Lecture 12

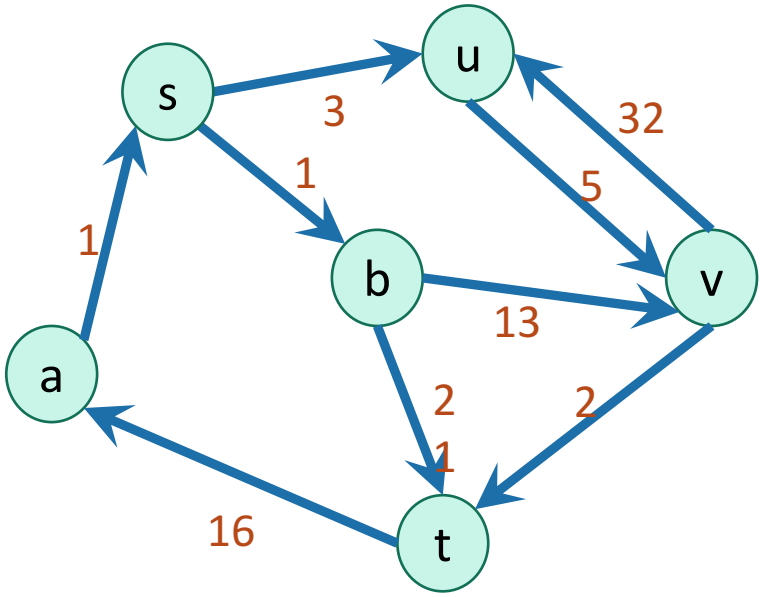
Bellman-Ford, Floyd-Warshall, and Dynamic Programming!

Announcements

- HW5 due Friday
- Midterms have been graded!
  - Pick up your exam **after class**.
  - Average: 84, Median: 87
  - Max: 100 (x4)
- I am very happy with how well y'all did!
- Regrade policy:
  - Write out a regrade request as you would on Gradescope.
  - Hand your exam and your request to **me** after class on Wednesday or in my office hours Tuesday (or by appointment).

Last time

- Dijkstra's algorithm!
- Solves single-source shortest path in weighted graphs.

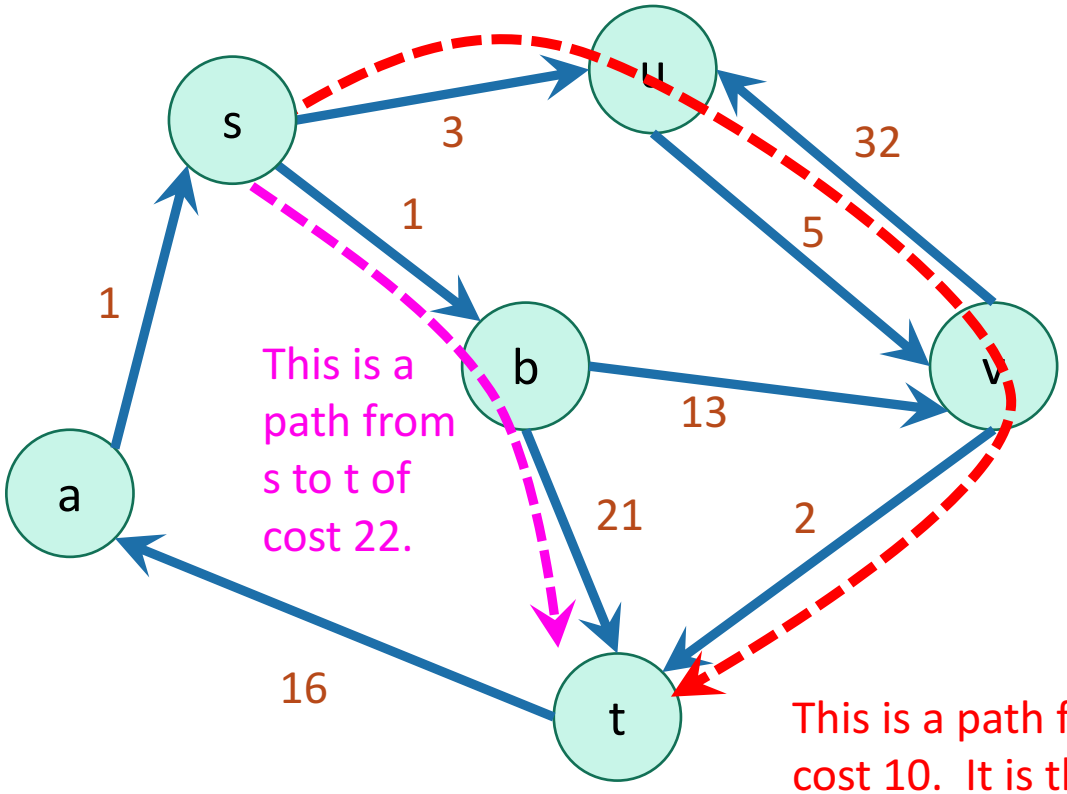


Today

- Bellman-Ford algorithm
  - Another **single-source shortest path algorithm**
- This is an example of **dynamic programming**
  - We'll see what that means
- Floyd-Warshall algorithm
  - An "all-pairs" shortest path algorithm
  - Another example of **dynamic programming**

Recall

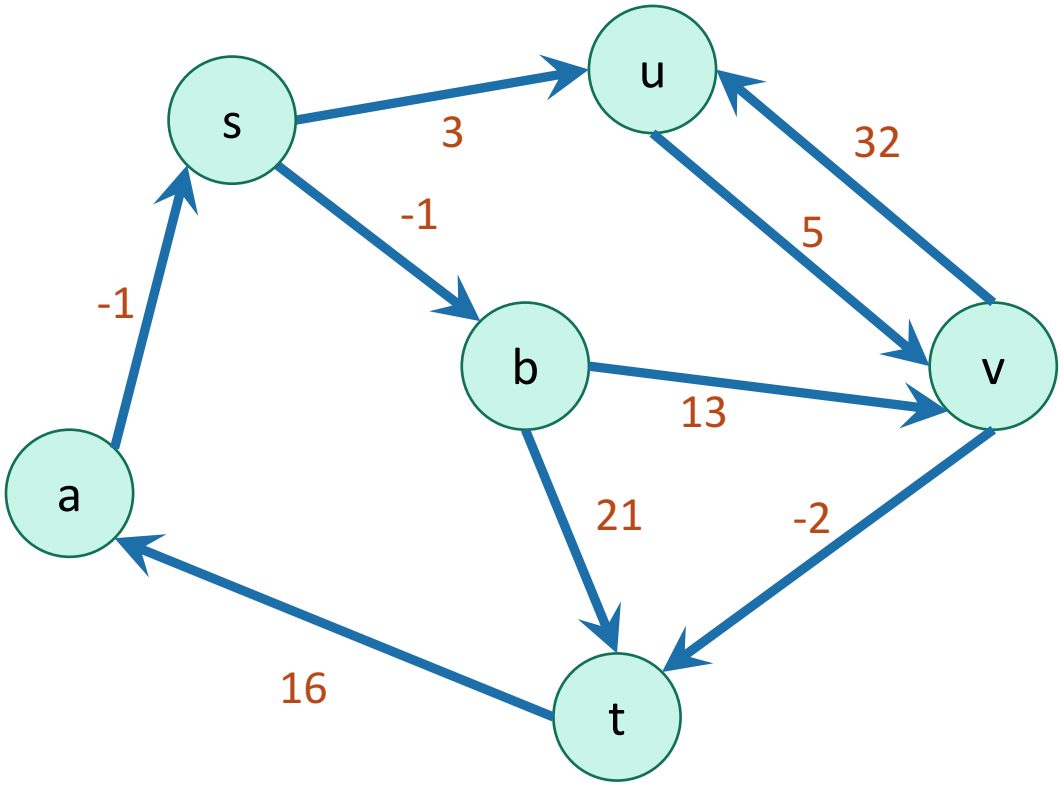
- A weighted directed graph:



- Weights on edges represent costs.
- The cost of a path is the sum of the weights along that path.
- A shortest path from s to t is a directed path from s to t with the smallest cost.
- The single-source shortest path problem is to find the shortest path from s to v for all v in the graph.

One drawback to Dijkstra

- Might not work with negative edge weights
  - On your homework!



Why would we ever have negative weights?

- Negative costs might mean benefits.
- eg, it costs me -$2 when I get $2.

Bellman-Ford Algorithm

- Slower (but arguably simpler) than Dijkstra's algorithm.
- Works with negative edge weights.

*We won't actually store all these, but let's pretend we do for now.

Bellman-Ford Algorithm

- We keep* an array $d^{(k)}$ of length n for each k = 0, 1, …, n-1.

Formally, we will maintain the loop invariant:

$d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges in it, for all b in V.

- For example, **this** is the shortest path from s to t with at most two edges in it.
- But it's not the shortest path from s to t (with any number of edges).
  - That's **this one**.

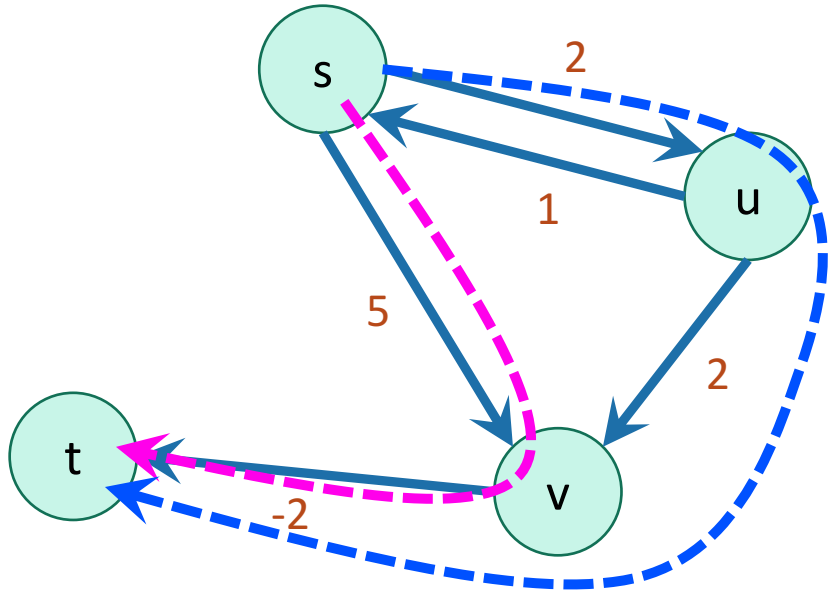
$d^{(0)}$

$d^{(1)}$

$d^{(2)}$

$d^{(3)}$

s u v t



*We won't actually store all these, but let's pretend we do for now.

Bellman-Ford Algorithm

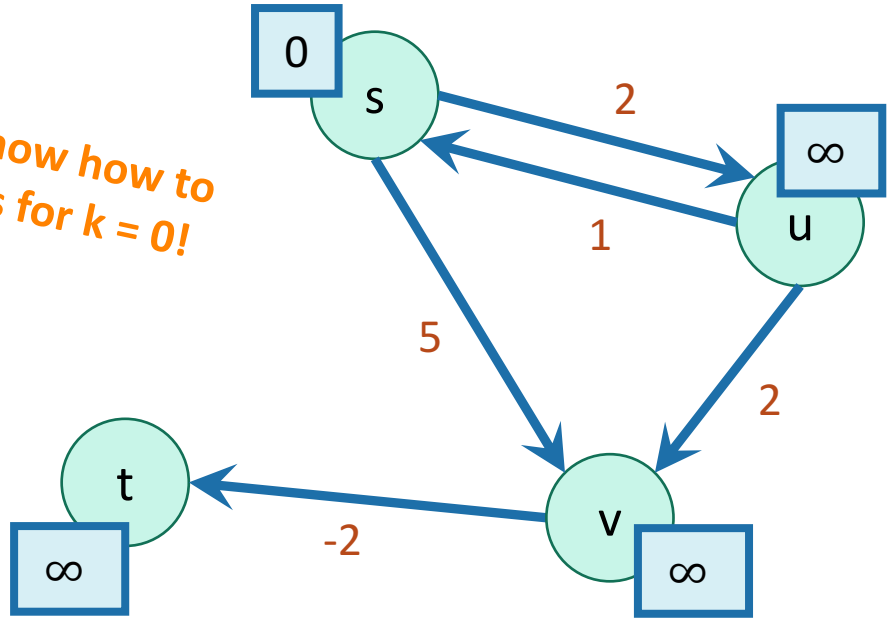
- We keep* an array $d^{(k)}$ of length n for each k = 0, 1, …, n-1.

Formally, we will maintain the loop invariant:

$d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges in it, for all b in V.

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$				
$d^{(2)}$				
$d^{(3)}$				

We know how to do this for k = 0!



Now update!

While maintaining:

$d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges in it.

- We will use the table $d^{(0)}$ to fill in $d^{(1)}$
- Then use $d^{(1)}$ to fill in $d^{(2)}$
- …
- Then use $d^{(k-1)}$ to fill in $d^{(k)}$
- …
- Then use $d^{(n-2)}$ to fill in $d^{(n-1)}$

This eventually gives us what we want:

- $d^{(k)}[a]$ is the shortest path from s to a with at most k edges.
- Eventually we'll get all the shortest paths…

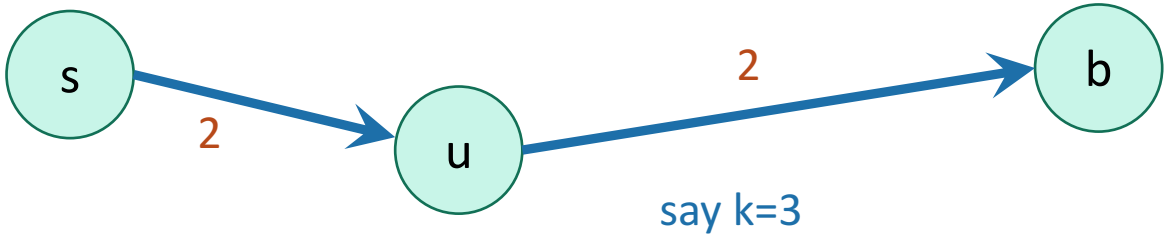
How do we get $d^{(k)}[b]$ from $d^{(k-1)}$?

- Two cases:

Want to maintain: $d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges in it.

Case 1: the shortest path from s to b with at most k edges actually has at most k-1 edges.

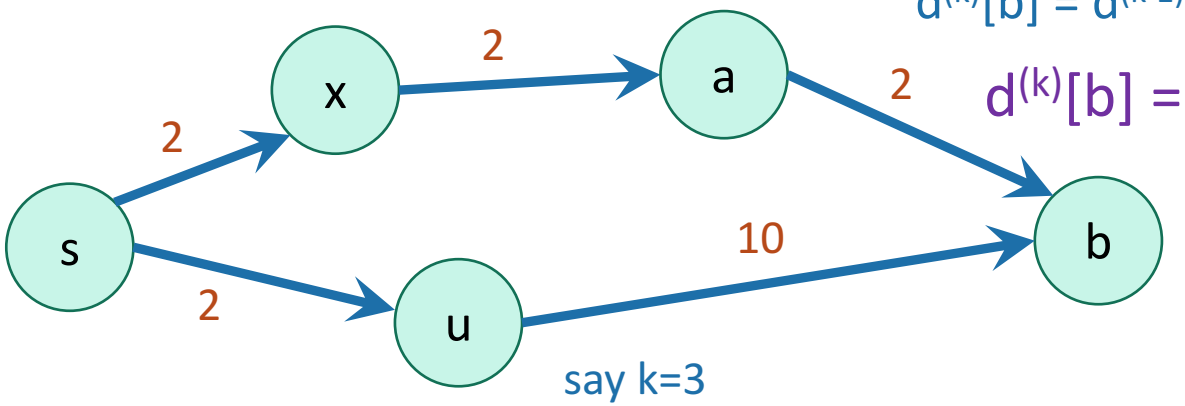
$$d^{(k)}[b] = d^{(k-1)}[b]$$



Case 2: the shortest path from s to b with at most k edges really has k edges.

$$d^{(k)}[b] = d^{(k-1)}[a] + w(a,b) \text{ for some a...}$$

$$d^{(k)}[b] = \min_a \{d^{(k-1)}[a] + w(a,b)\}$$



Bellman-Ford Algorithm*

- Bellman-Ford*(G,s):
  - Initialize $d^{(k)}$ for k = 0, …, n-1
  - $d^{(0)}[v] = \infty$ for all v other than s
  - $d^{(0)}[s] = 0$.
  - For k = 1, …, n-1:
    - For b in V:
      - $d^{(k)}[b] \leftarrow \min\{ d^{(k-1)}[b], \min_a \{d^{(k-1)}[a] + \text{weight}(a,b)\} \}$
  - Return $d^{(n-1)}$

This minimum is over all a so that (a,b) is in E

Case 1

Case 2

If we set $d^{(k)}[b]$ to be the minimum of the previous two cases, then we maintain the loop invariant that:

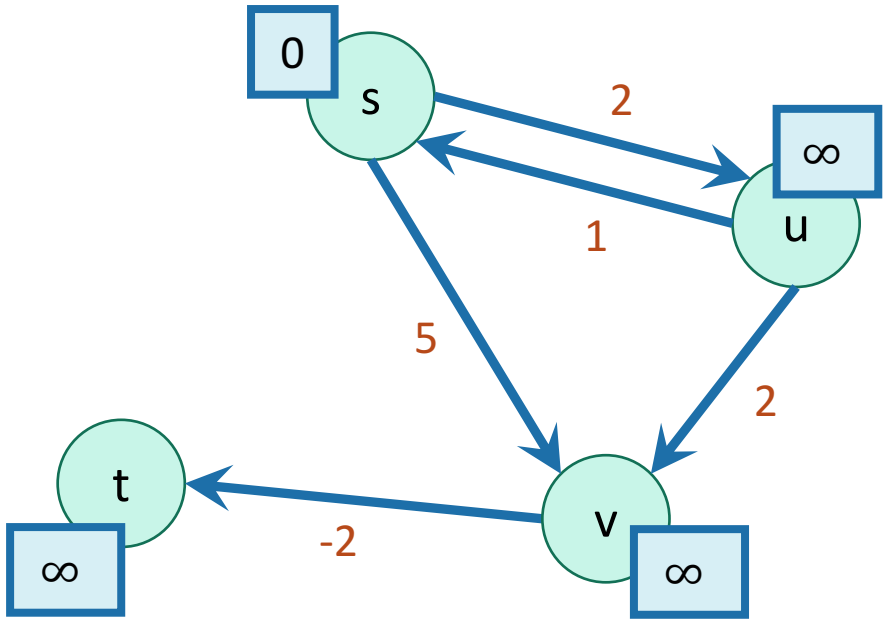
$d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges in it.

Bellman-Ford Algorithm* Example

$d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges in it.

- For k = 1,…,n-1:
  - For b in V:
    - $d^{(k)}[b] \leftarrow \min\{ d^{(k-1)}[b], \min_a \{d^{(k-1)}[a] + \text{weight}(a,b)\} \}$

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$				
$d^{(2)}$				
$d^{(3)}$				

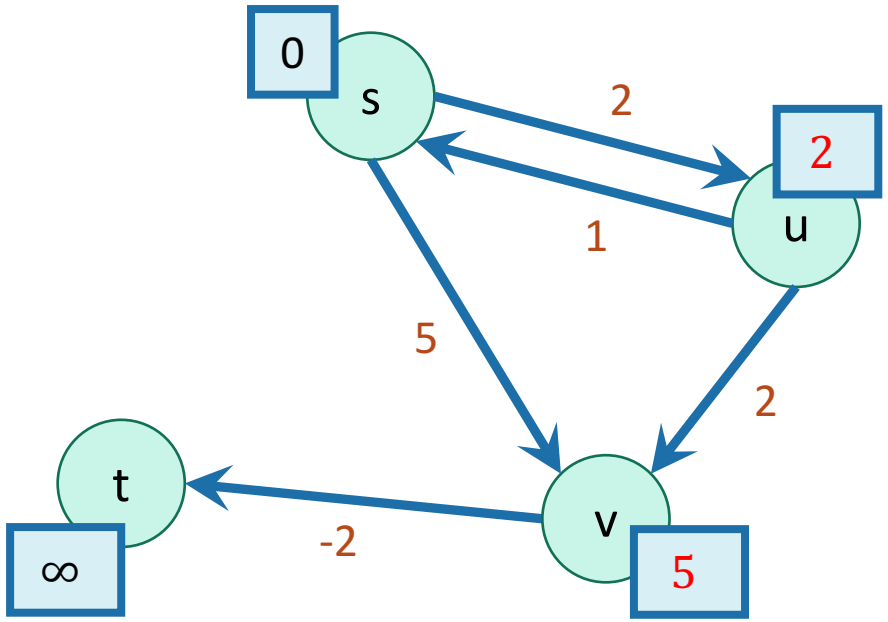


Bellman-Ford Algorithm* Example

$d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges in it.

- For k = 1,…,n-1:
  - For b in V:
    - $d^{(k)}[b] \leftarrow \min\{ d^{(k-1)}[b], \min_a \{d^{(k-1)}[a] + \text{weight}(a,b)\} \}$

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$				
$d^{(3)}$				

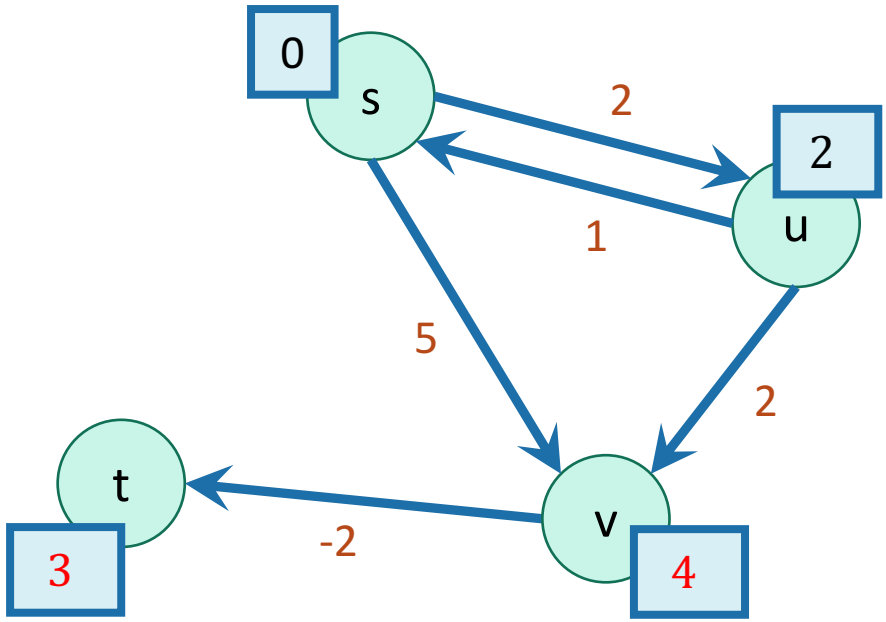


Bellman-Ford Algorithm* Example

$d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges in it.

- For k = 1,…,n-1:
  - For b in V:
    - $d^{(k)}[b] \leftarrow \min\{ d^{(k-1)}[b], \min_a \{d^{(k-1)}[a] + \text{weight}(a,b)\} \}$

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$				

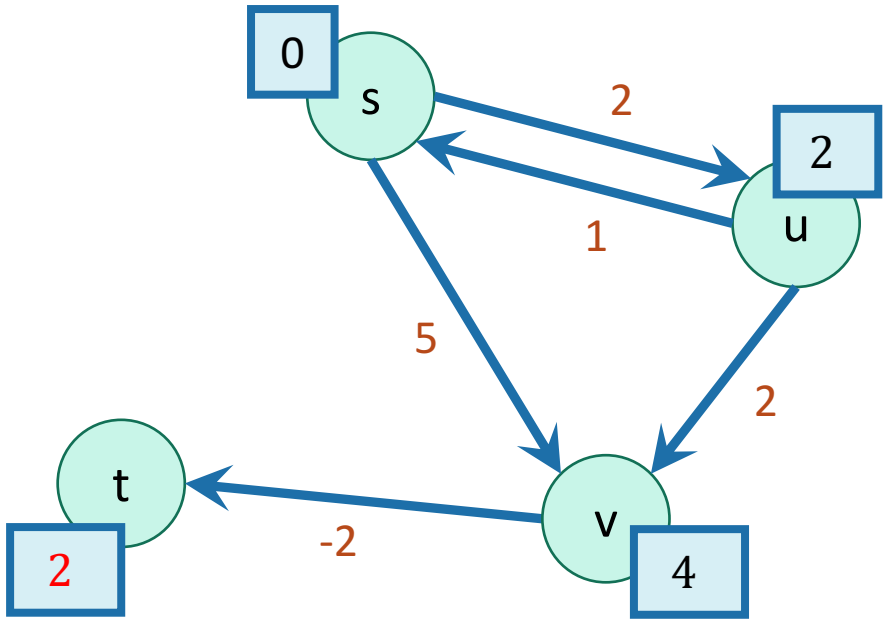


Bellman-Ford Algorithm* Example

$d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges in it.

- For k = 1,…,n-1:
  - For b in V:
    - $d^{(k)}[b] \leftarrow \min\{ d^{(k-1)}[b], \min_a \{d^{(k-1)}[a] + \text{weight}(a,b)\} \}$

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0	2	4	2



Bellman-Ford Algorithm* Example

$d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges in it.

SANITY CHECK:

- The shortest path with 1 edge from s to t has cost ∞. (there is no such path).
- The shortest path with 2 edges from s to t has cost 3. (s-v-t)
- The shortest path with 3 edges from s to t has cost 2. (s-u-v-t)

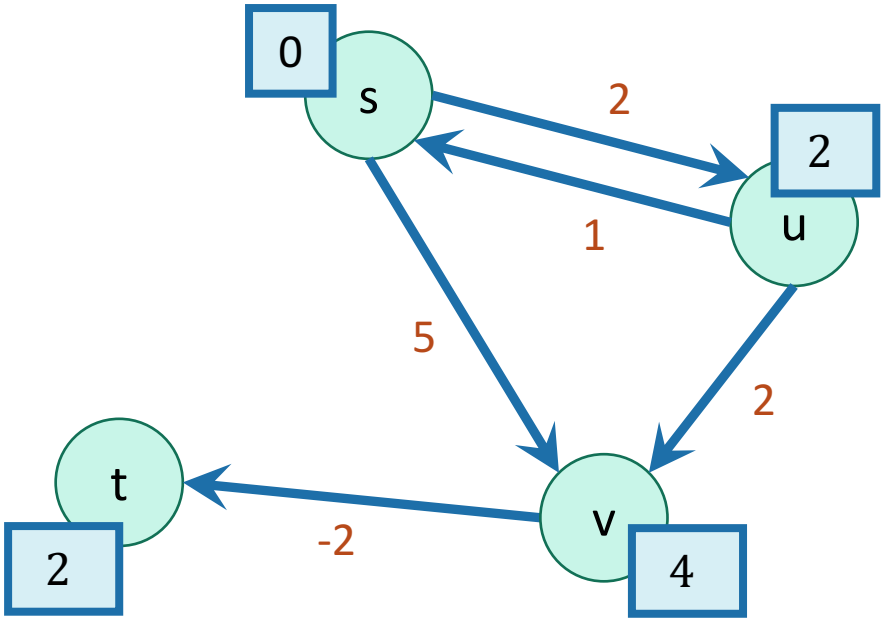
And this one is the shortest path!!!

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞

	s	u	v	t
$d^{(1)}$	0	2	5	∞

	s	u	v	t
$d^{(2)}$	0	2	4	3

	s	u	v	t
$d^{(3)}$	0	2	4	2



How do we actually implement this?

(This is what the * on all the previous slides was for).

- Don't actually keep all the arrays $d^{(k)}$ around.
  - Just keep two of them at a time, that's all we need.
- Running time: O(mn)
  - That's worse than Dijkstra, but BF can handle negative edge weights.
- Space complexity:
  - We need space to store the graph and two arrays of size n.

***WARNING:** This is slightly different from the version of Bellman-Ford in CLRS. But we will stick with what we just saw for pedagogical reasons.

See Lecture Notes 11.5 (listed on the webpage in the Lecture 12 box) for notes on the analysis of the slightly different CLRS version.

Bellman-Ford Algorithm*

Let's stick with this version though.

- Bellman-Ford*(G,s):
  - Initialize $d^{(k)}$ for k = 0, …, n-1
  - $d^{(0)}[v] = \infty$ for all v other than s
  - $d^{(0)}[s] = 0$.
  - For k = 1, …, n-1:
    - For b in V:
      - $d^{(k)}[b] \leftarrow \min\{ d^{(k-1)}[b], \min_a \{d^{(k-1)}[a] + \text{weight}(a,b)\} \}$
  - Return $d^{(n-1)}$

Why does it work?

- First, we've been asserting that:

> $d^{(n-1)}[b]$ is the cost of the shortest path from s to b with at most n-1 edges in it.

- Technically, this requires proof!
  - We've basically already seen the proof!
  - It follows from induction with the inductive hypothesis

> $d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges in it.

Work out the details of this proof on your own! To help you, there's an outline on the next slide. (Which we'll skip now).

Sketch of proof [skip this in lecture]

that this thing we've been asserting is really true

- Inductive hypothesis: $d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges in it.
- Base case: For k = 0:

0	∞	∞	∞

- Inductive step:
  - $d^{(k)}[b] \leftarrow \min\{ d^{(k-1)}[b], \min_a \{d^{(k-1)}[a] + \text{weight}(a,b)\} \}$

Case 2: the shortest path from s to b of length at most k edges has exactly k edges

Case 1: the shortest path from s to be has <k edges

  - In either case, we make the correct update.
- Conclusion:

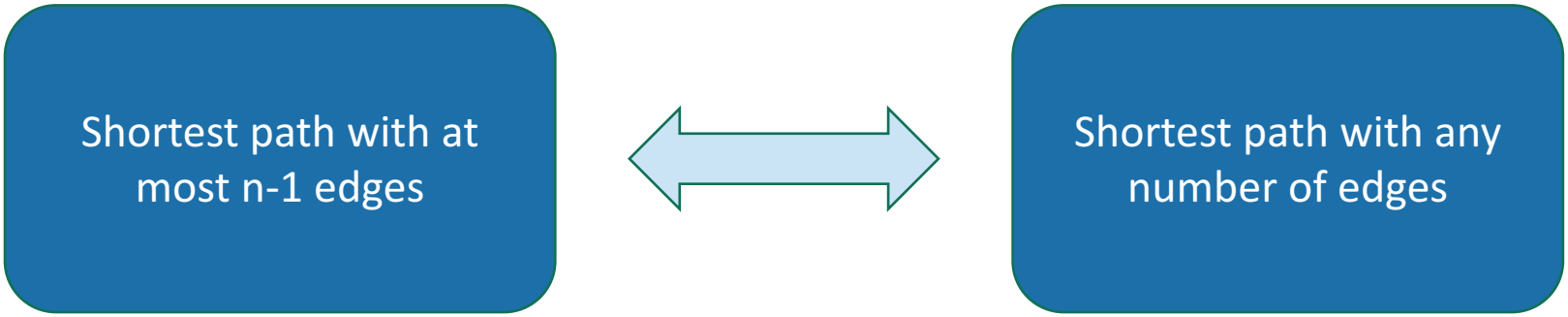
When k = n-1, the inductive hypothesis reads: $d^{(n-1)}[b]$ is the cost of the shortest path from s to b with at most n-1 edges in it.

Is this the conclusion we want?

$d^{(n-1)}[b]$ is the cost of the shortest path from s to b with at most n-1 edges in it.

- We still need to prove that this implies BF* is correct.
  - We return $d^{(n-1)}$
  - Need to show $d^{(n-1)}[a]$ = distance(s,a).

- Enough to show:

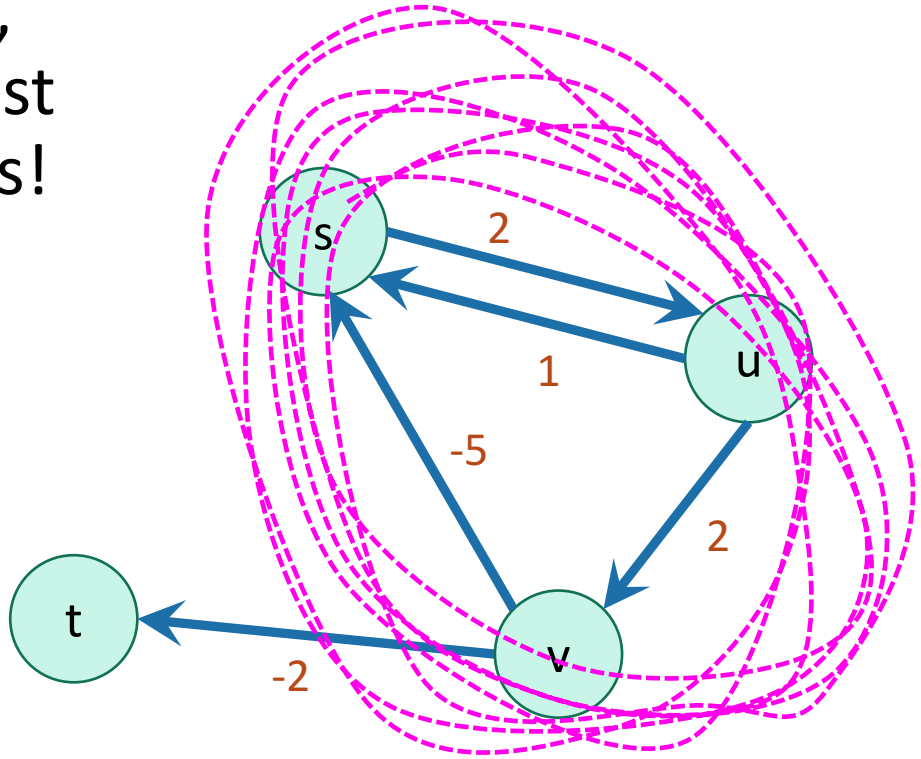


DANGER!

Shortest path with at most n-1 edges ⟺ Shortest path with any number of edges

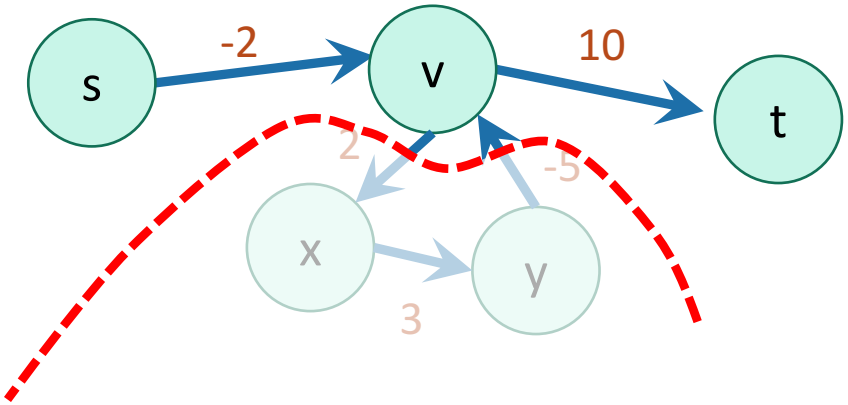
- If the graph has a negative cycle, this might not be true.
- If there is a negative cycle, there may not be a shortest path between two vertices!

A negative cycle is a directed cycle with negative total cost



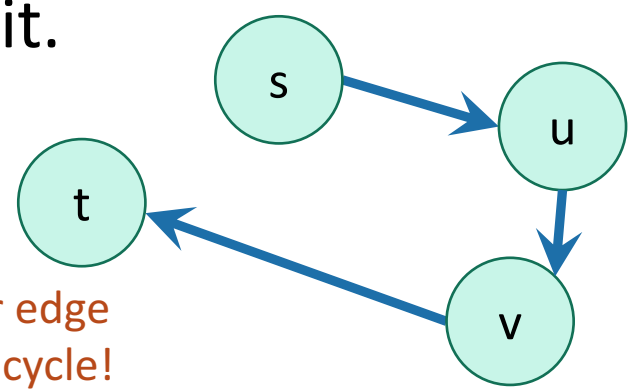
But if there is no negative cycle

- Then not only are there shortest paths, but actually there's always a simple shortest path.



This cycle isn't helping. Just get rid of it.

- A simple path in a graph with n vertices has at most n-1 edges in it.



Can't add another edge without making a cycle!

"Simple" means that the path has no cycles in it.

Let's go after a new conclusion.

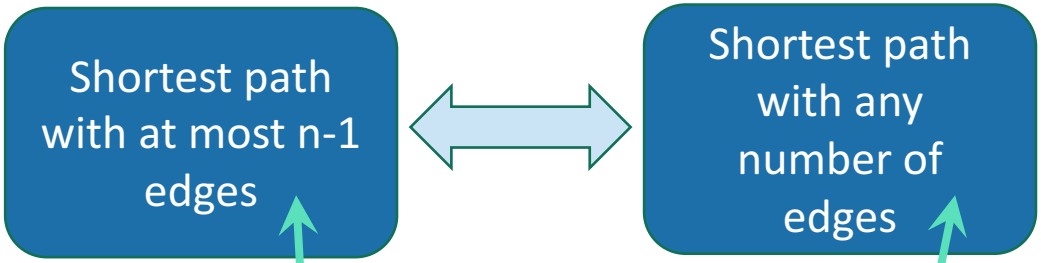
- Theorem:
  - The Bellman-Ford Algorithm* is correct as long as G has no negative cycles.

*We will prove this for our version of Bellman-Ford.
See Notes 11.5 or CLRS for CLRS version.

Proof

- By induction, $d^{(n-1)}[b]$ is the cost of the shortest path from s to b with at most n-1 edges in it.

- If there are no negative cycles,



  - This is because the shortest path is WLOG simple, and all simple paths have at most n-1 edges.

- So the thing we return is equal to the thing we want to return.

So that proves:

- Theorem:
  - The Bellman-Ford Algorithm* is correct as long as G has no negative cycles.
  - Further, if G has a negative cycle, Bellman-Ford can detect that.
    - (See Notes 11.5)

What have we learned?

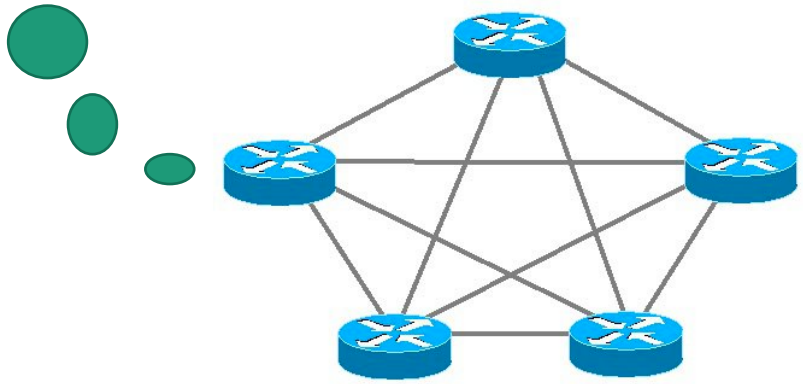
- The Bellman-Ford algorithm is slower than Dijkstra:
  - O(mn) time
- But it works with negative edges weights.
  - You'll see how Dijkstra does with negative edge weights in HW5.
- It doesn't work with negative cycles, but in that case shortest paths don't even make sense.

Bellman-Ford is also used in practice.

- eg, Routing Information Protocol (RIP) uses something like Bellman-Ford.
  - Older protocol, not used as much anymore.

- Each router keeps a **table** of distances to every other router.
- Periodically we do a Bellman-Ford update.
- This also means that if there are changes in the network, this will propagate. (maybe slowly…)

Destination	Cost to get there	Send to whom?
172.16.1.0	34	172.16.1.1
10.20.40.1	10	192.168.1.2
10.155.120.1	9	10.13.50.0



This was an example of…

What is dynamic programming?

- It is an algorithm design paradigm
  - like divide-and-conquer is an algorithm design paradigm.
- Usually it is for solving optimization problems
  - eg, ***shortest*** path

Elements of dynamic programming

- Big problems break up into little problems.
  - eg, Shortest path with at most k edges.
- The optimal solution of a problem can be expressed in terms of optimal solutions of smaller sub-problems.
  - eg, $d^{(k)}[b] \leftarrow \min\{ d^{(k-1)}[b], \min_a \{d^{(k-1)}[a] + \text{weight}(a,b)\} \}$

We call this "optimal sub-structure"

Elements of dynamic programming II

- The sub-problems overlap a lot.
  - eg, Lots of different entries of $d^{(k)}$ ask for $d^{(k-1)}[a]$.
  - This means that we can save time by solving a sub-problem just once and storing the answer.

We call this "overlapping sub-problems"

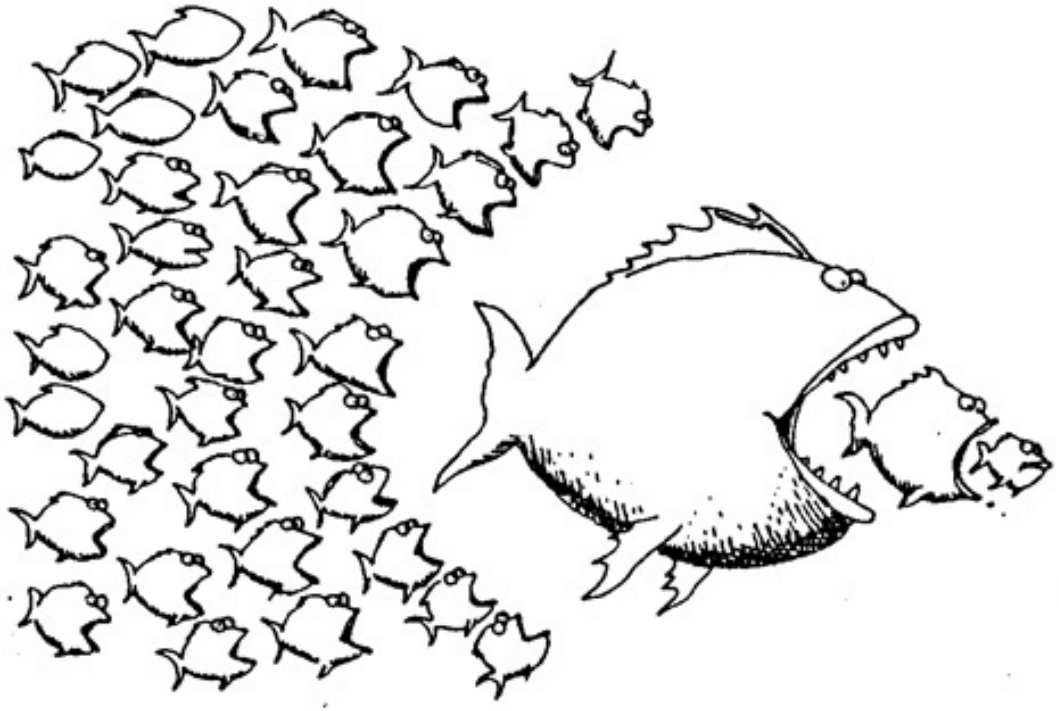
Elements of dynamic programming III

- Optimal substructure.
  - Optimal solutions to sub-problems are sub-solutions to the optimal solution of the original problem.
- Overlapping subproblems.
  - The subproblems show up again and again

- Using these properties, we can design a dynamic programming algorithm:
  - Keep a table of solutions to the smaller problems.
  - Use the solutions in the table to solve bigger problems.
  - At the end we can use information we collected along the way to find the optimal solution.
    - eg, recover the shortest path (not just its cost).

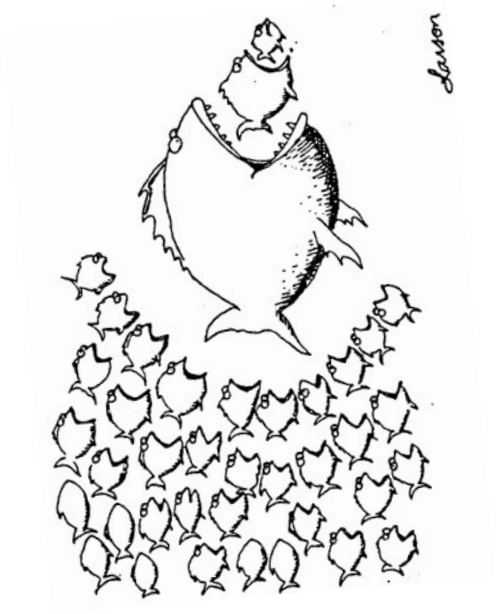
Two ways to think about and/or implement DP algorithms

- Top down
- Bottom up



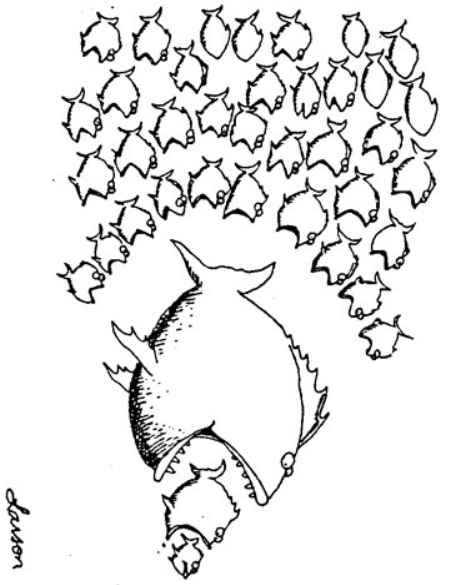
This picture isn't hugely relevant but I like it. Larson

Bottom up approach



- What we just saw.
- Solve the small problems first
  - fill in $d^{(0)}$
- Then bigger problems
  - fill in $d^{(1)}$
- …
- Then bigger problems
  - fill in $d^{(n-2)}$
- Then finally solve the real problem.
  - fill in $d^{(n-1)}$

Top down approach



- Think of it like a recursive algorithm.
- To solve the big problem:
  - Recurse to solve smaller problems
    - Those recurse to solve smaller problems
      - etc..

- The difference from divide and conquer:
  - Memo-ization
  - Keep track of what small problems you've already solved to prevent re-solving the same problem twice.

Example: top-down** version of BF*

The actual pseudocode here isn't important, I just want to talk about the structure of it.

- Bellman-Ford*(G,s):
  - Initialize a bunch of empty tables $d^{(k)}$ for k=0,…,n-1,
  - Fill in $d^{(0)}$
  - **for** b in V:
    - BF*_helper(G, s, b, n-1)

If we don't already have $d^{(k-1)}[a]$ stored, **compute it**. But store it so that we never have to do that again!

- BF*_helper(G, s, b, k):
  - **For each** a so that (a,b) in E, and also for a=b:
    - **If** $d^{(k-1)}[a]$ is not already in the table:
      - $d^{(k-1)}[a]$ = BF*_helper(G, s, a, k-1)
  - **return** $\min\{ d^{(k-1)}[b], \min_a \{d^{(k-1)}[a] + \text{weight}(a,b)\} \}$

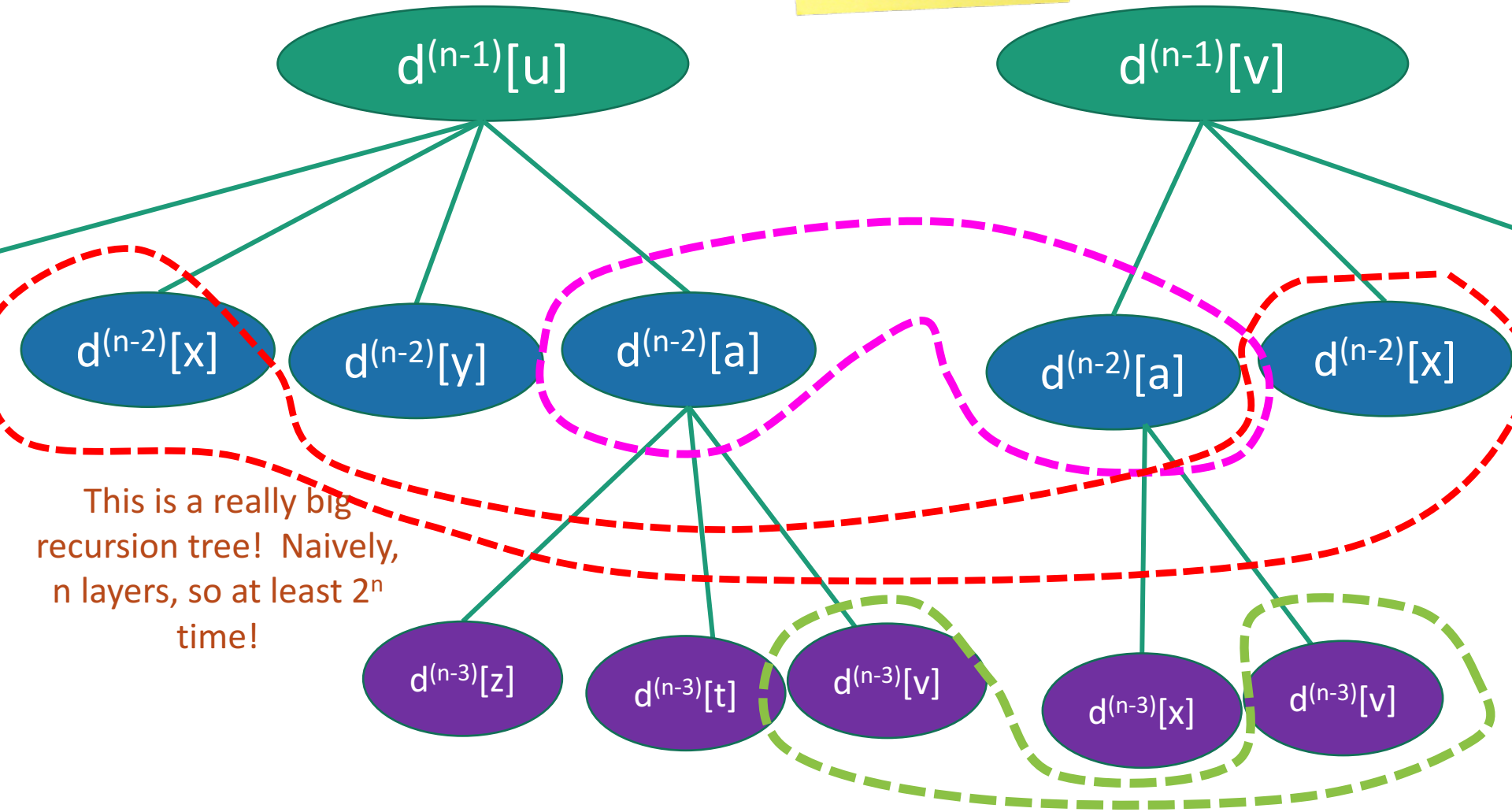
*Not the actual Bellman-Ford algorithm; we don't want to keep all these tables around

**Probably not the best way to think about Bellman-Ford: this is for DP pedagogy only!

Visualization

top-down approach

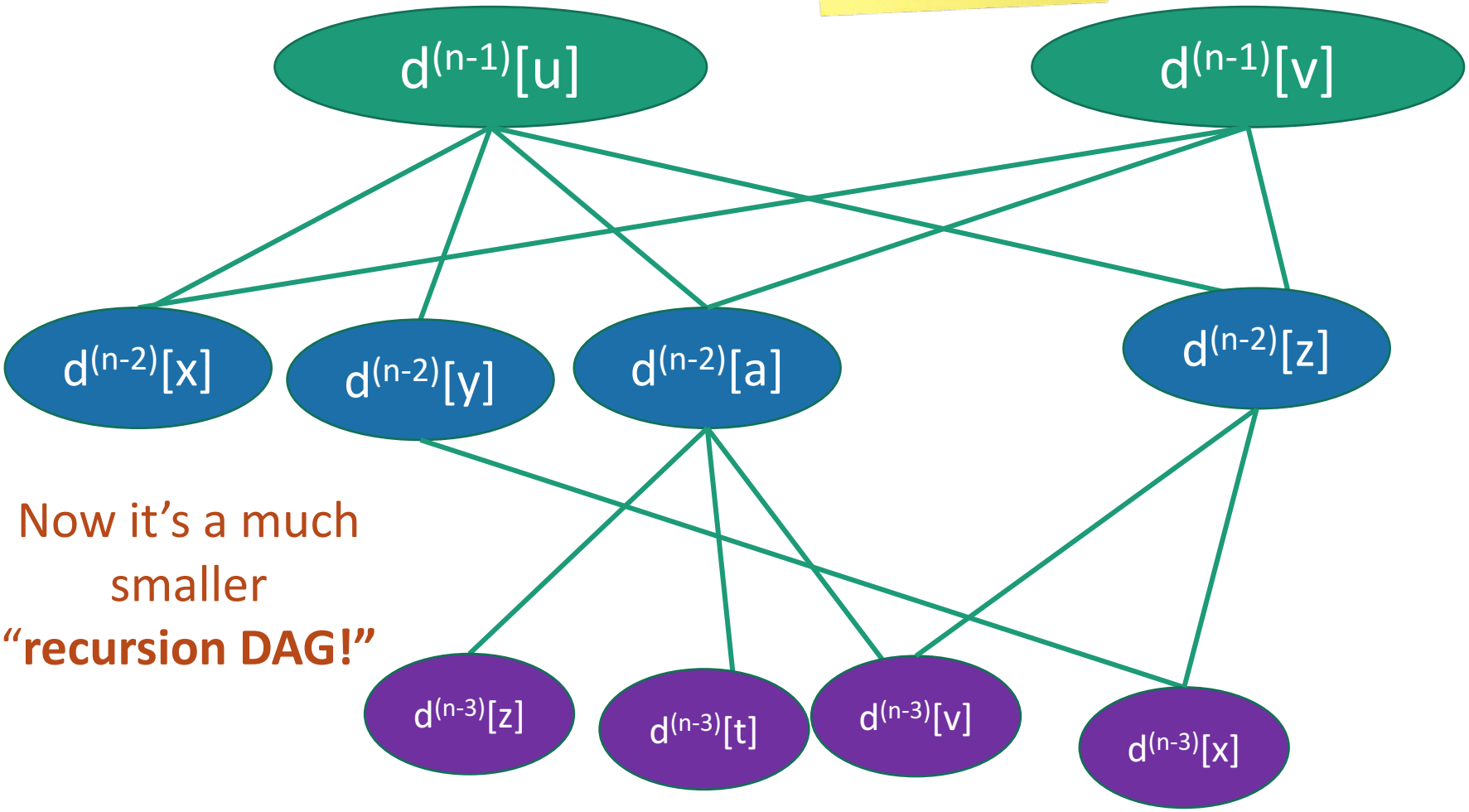
Identify repeated nodes and don't do the same work twice!



This is a really big recursion tree! Naively, n layers, so at least 2^n time!

Visualization
top-down approach

Identify repeated nodes and don't do the same work twice!



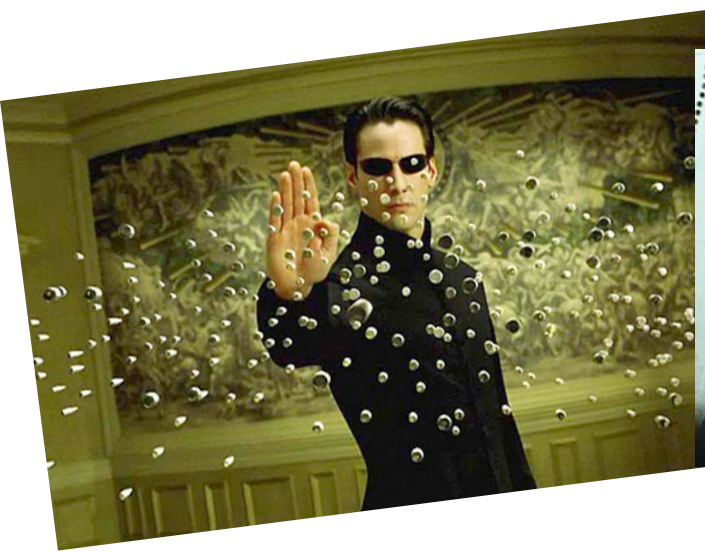
Now it's a much smaller **"recursion DAG!"**

What have we learned?

- ***Dynamic programming:***
  - Paradigm in algorithm design.
  - Useful when there's **optimal substructure:**
    - optimal solutions to a big problem break up in to optimal sub-solutions of subproblems.
  - Useful when there are **overlapping subproblems**:
    - Use memo-ization (aka, put it in a table) to prevent repeated work.
  - Can be implemented **bottom-up** or **top-down**.
  - It's a fancy name for a pretty common-sense idea:
    - **Don't duplicate work if you don't have to!**

Why "dynamic programming" ?

- Programming refers to finding the optimal "program."
  - as in, a shortest route is a *plan* aka a *program*.
- Dynamic refers to the fact that it's multi-stage.
- But also it's just a fancy-sounding name.



Manipulating computer code in an action movie?

Why "dynamic programming" ?

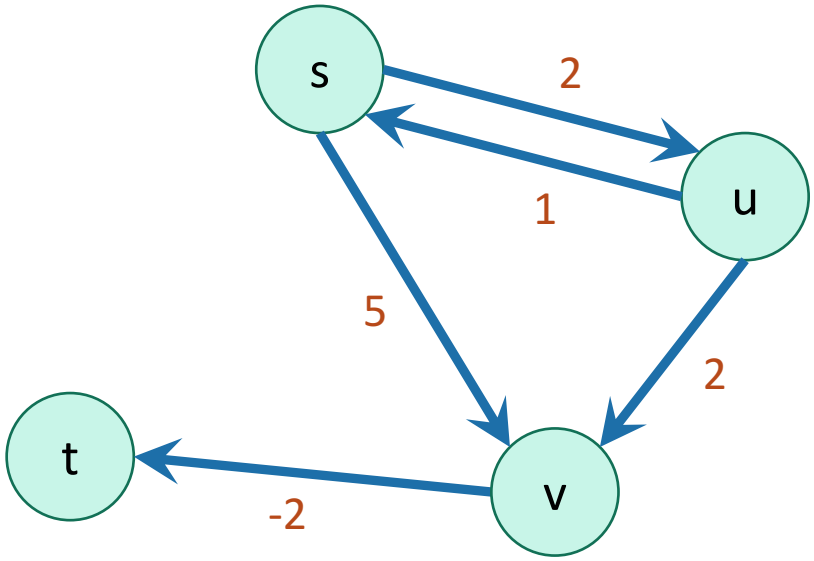
- Richard Bellman invented the name in the 1950's.
- At the time, he was working for the RAND Corporation, which was basically working for the Air Force, and government projects needed flashy names to get funded.
- From Bellman's autobiography:
  - "It's impossible to use the word, dynamic, in the pejorative sense…I thought dynamic programming was a good name. It was something not even a Congressman could object to."

Another example

- **Floyd-Warshall Algorithm**
- This is an algorithm for All-Pairs Shortest Paths (APSP)
  - That is, I want to know the shortest path from u to v for **ALL pairs** u,v of vertices in the graph.
  - Not just from a special single source s.

Destination

Source	s	u	v	t
s	0	2	4	2
u	1	0	2	0
v	∞	∞	0	-2
t	∞	∞	∞	0



Another example

- **Floyd-Warshall Algorithm**
- This is an algorithm for All-Pairs Shortest Paths (APSP)
  - That is, I want to know the shortest path from u to v for **ALL pairs** u,v of vertices in the graph.
  - Not just from a special single source s.
- Naïve solution (if we want to handle negative edge weights):
  - For all s in G:
    - Run Bellman-Ford on G starting at s.
  - Time $O(n \cdot nm) = O(n^2 m)$,
    - may be as bad as n^4 if $m=n^2$

Can we do better?

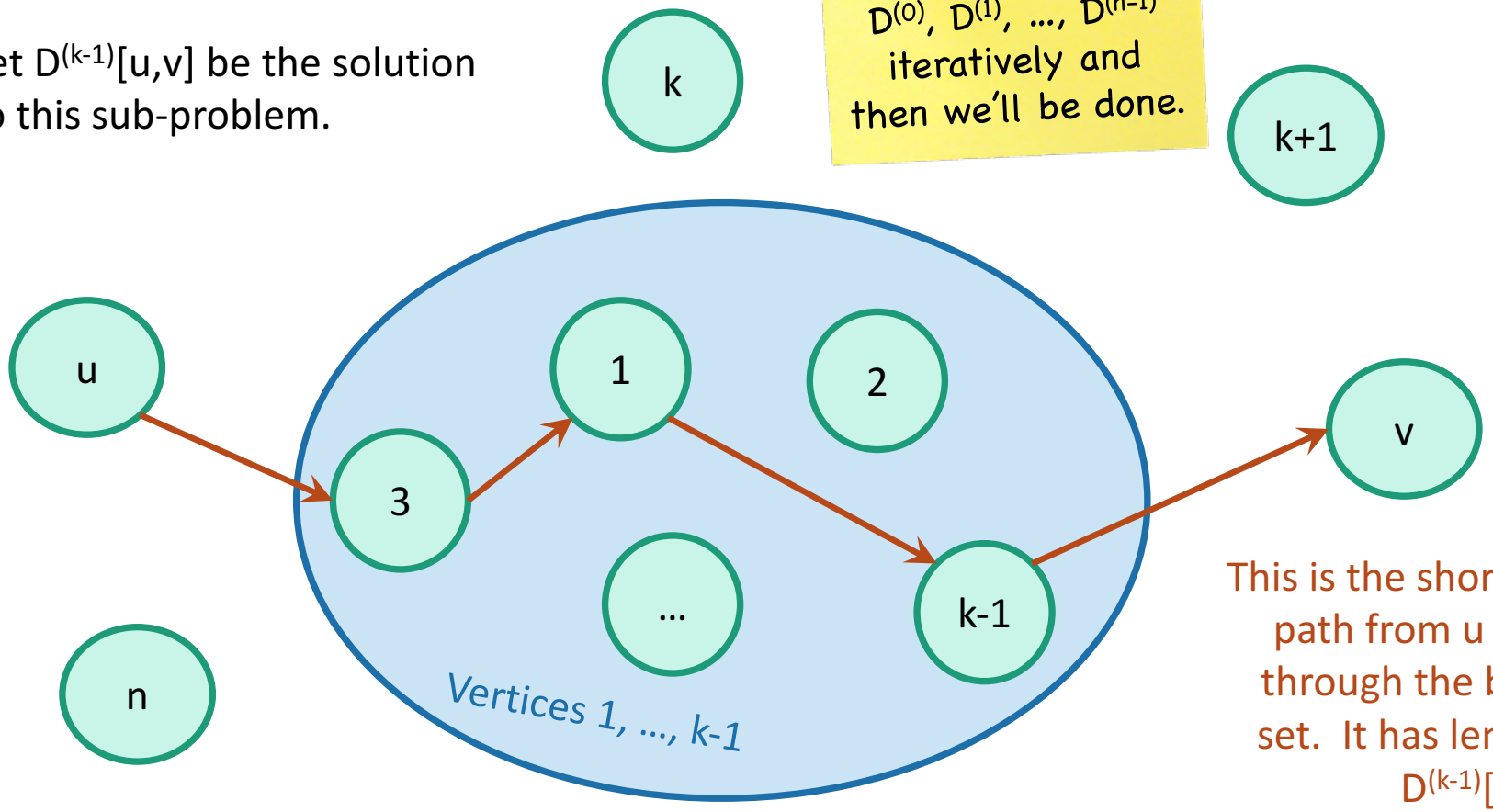
Optimal substructure

Label the vertices 1,2,…,n
(We omit edges in the picture below).

Sub-problem: For all pairs, u,v, find the cost of the shortest path from u to v, so that all the internal vertices on that path are in {1,…,k-1}.

Let $D^{(k-1)}[u,v]$ be the solution to this sub-problem.

Our DP algorithm will fill in the n-by-n arrays $D^{(0)}$, $D^{(1)}$, …, $D^{(n-1)}$ iteratively and then we'll be done.



This is the shortest path from u to v through the blue set. It has length $D^{(k-1)}[u,v]$

Optimal substructure

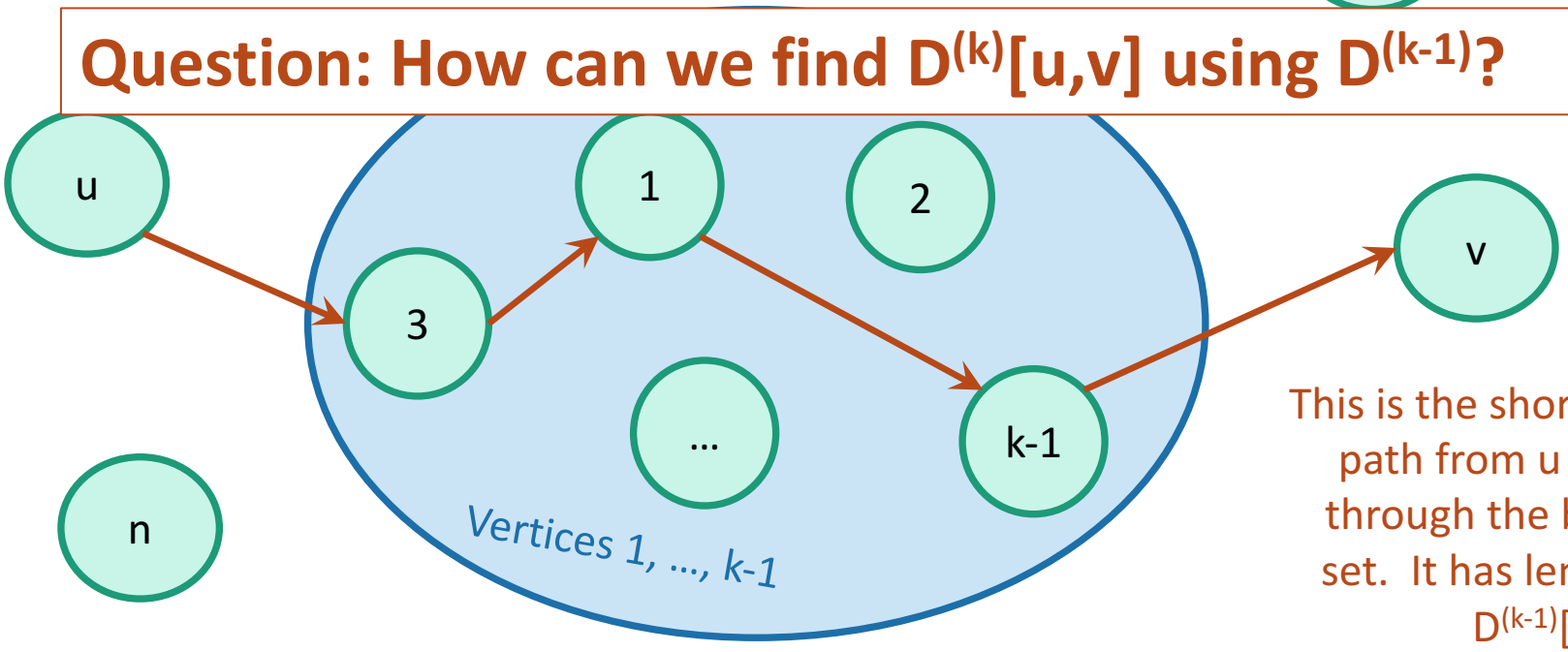
Label the vertices 1,2,…,n
(We omit edges in the picture below).

Sub-problem: For all pairs, u,v, find the cost of the shortest path from u to v, so that all the internal vertices on that path are in {1,…,k-1}.

Let $D^{(k-1)}[u,v]$ be the solution to this sub-problem.

Our DP algorithm will fill in the n-by-n arrays $D^{(0)}$, $D^{(1)}$, …, $D^{(n-1)}$ iteratively and then we'll be done.

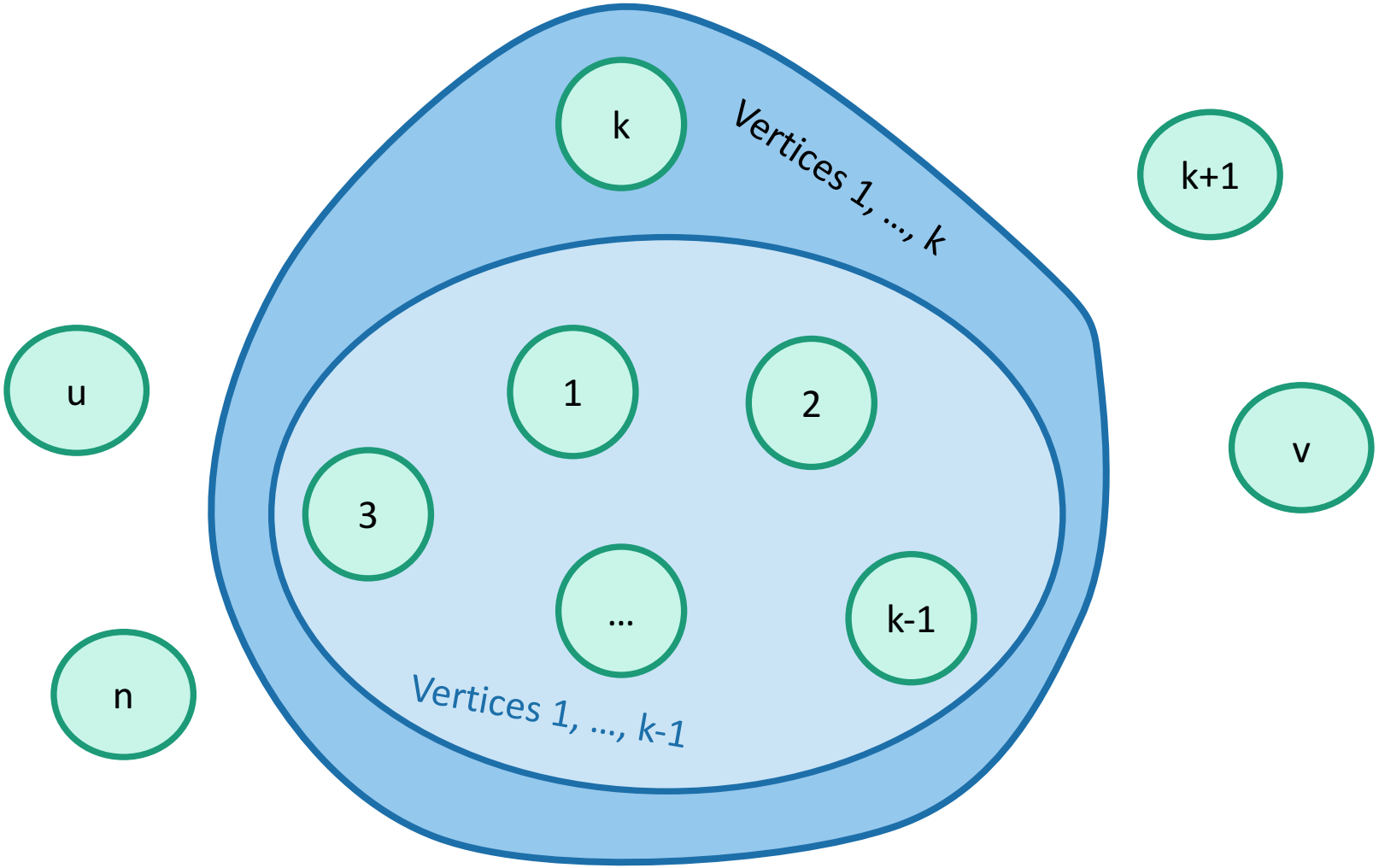
Question: How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?



This is the shortest path from u to v through the blue set. It has length $D^{(k-1)}[u,v]$

How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

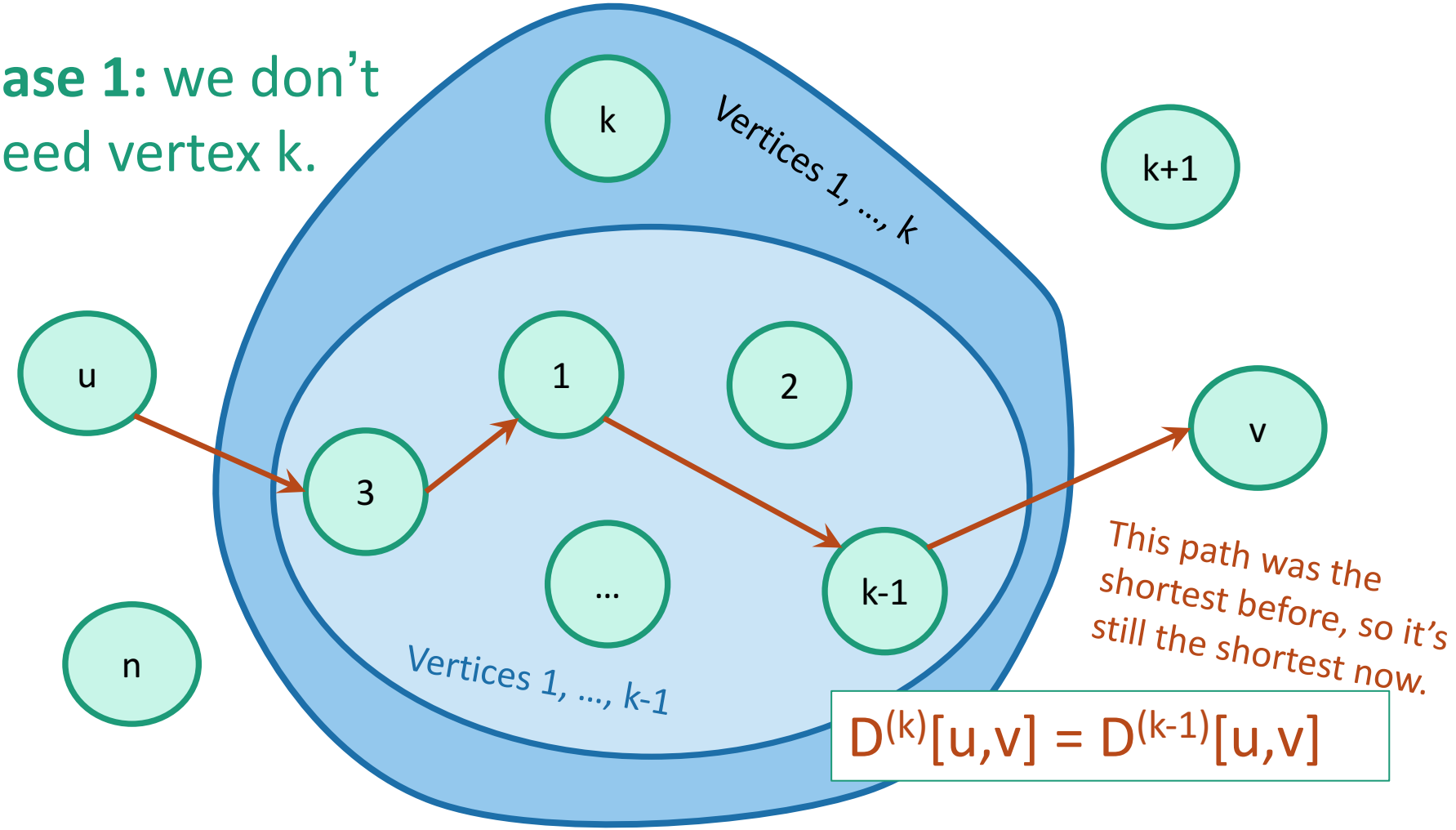
$D^{(k)}[u,v]$ is the cost of the shortest path from u to v so that all internal vertices on that path are in {1, …, k}.



How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

$D^{(k)}[u,v]$ is the cost of the shortest path from u to v so that all internal vertices on that path are in {1, …, k}.

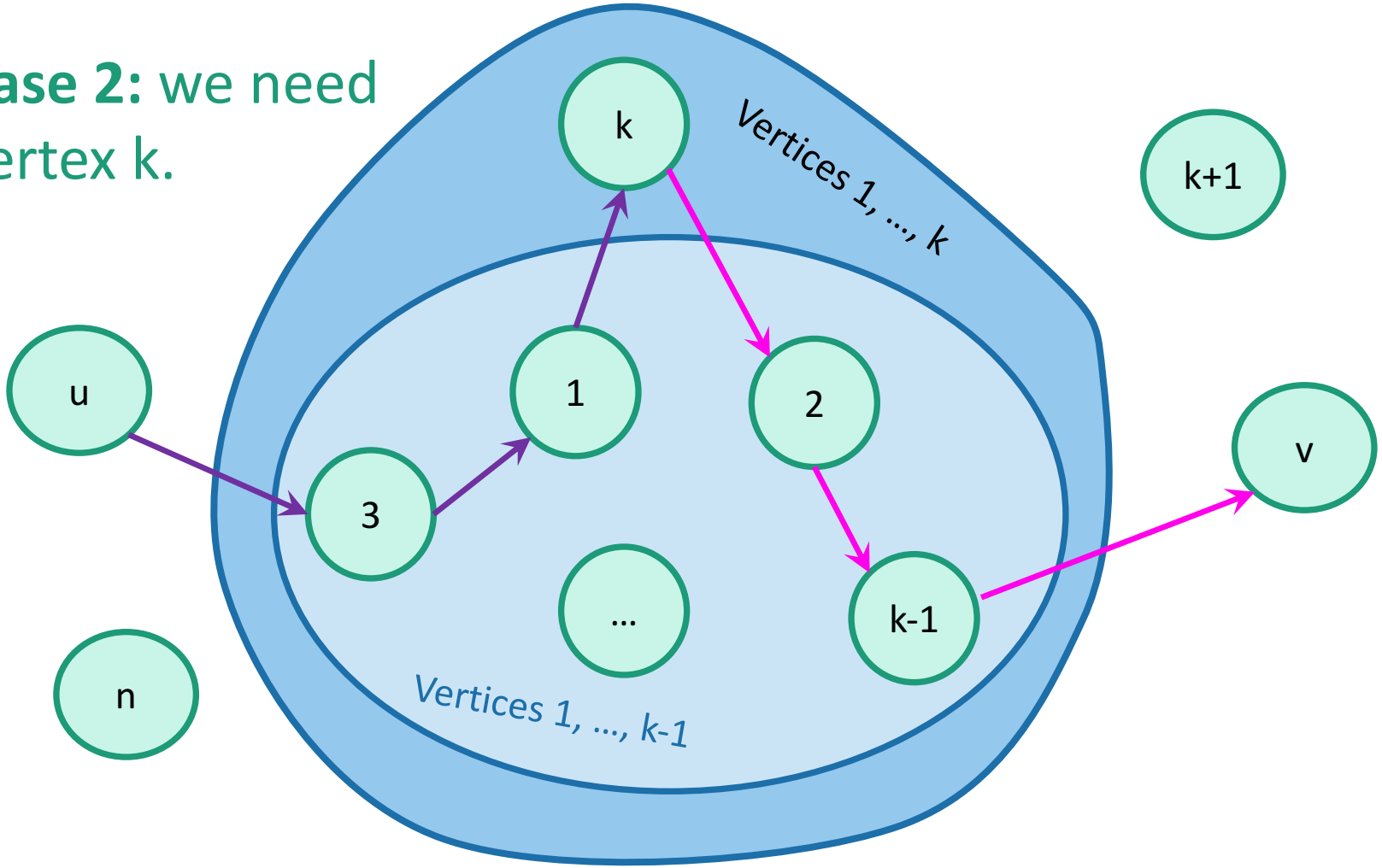
Case 1: we don't need vertex k.



How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

$D^{(k)}[u,v]$ is the cost of the shortest path from u to v so that all internal vertices on that path are in {1, …, k}.

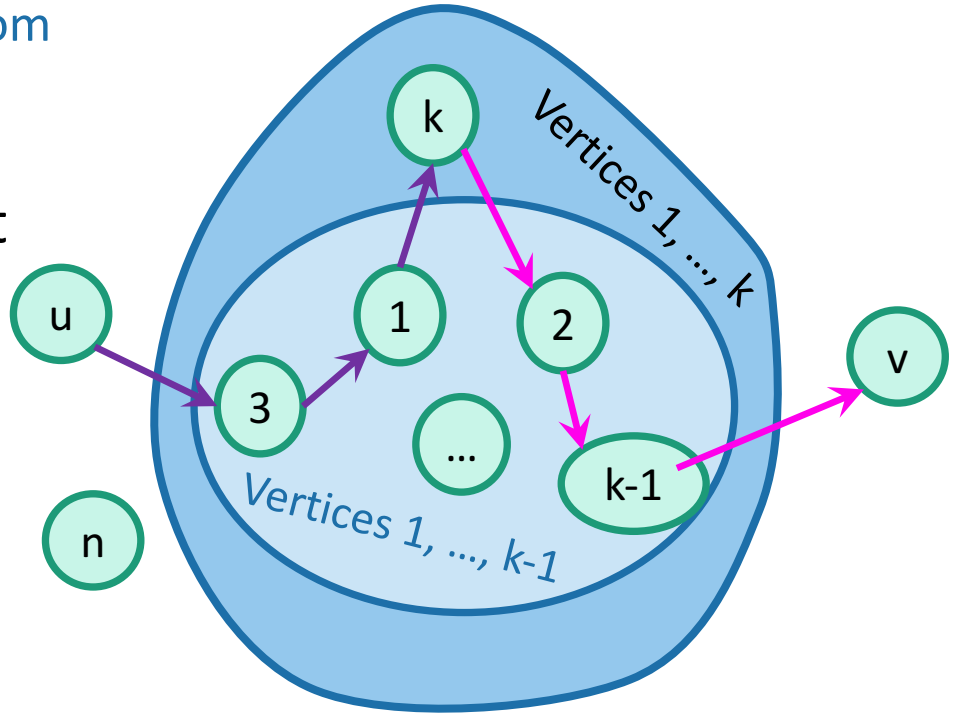
Case 2: we need vertex k.



Case 2 continued

- Suppose there are no negative cycles.
  - Then WLOG the shortest path from u to v through {1,…,k} is **simple**.
- If **that path** passes through k, it must look like this: →
- **This path** is the shortest path from u to k through {1,…,k-1}.
  - sub-paths of shortest paths are shortest paths
- Same for **this path**.

Case 2: we need vertex k.



$$D^{(k)}[u,v] = D^{(k-1)}[u,k] + D^{(k-1)}[k,v]$$

How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

- $D^{(k)}[u,v] = \min\{ D^{(k-1)}[u,v], D^{(k-1)}[u,k] + D^{(k-1)}[k,v] \}$

Case 1: Cost of shortest path through $\{1,\ldots,k-1\}$

Case 2: Cost of shortest path from u to k and then from k to v through $\{1,\ldots,k-1\}$

- Optimal substructure:
  - We can solve the big problem using smaller problems.
- Overlapping sub-problems:
  - $D^{(k-1)}[k,v]$ can be used to help compute $D^{(k)}[u,v]$ for lots of different u's.

How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

- $D^{(k)}[u,v] = \min\{ D^{(k-1)}[u,v], D^{(k-1)}[u,k] + D^{(k-1)}[k,v] \}$

Case 1: Cost of shortest path through {1,…,k-1}

Case 2: Cost of shortest path from u to k and then from k to v through {1,…,k-1}

- Using our ***Dynamic programming*** paradigm, this immediately gives us an algorithm!

Floyd-Warshall algorithm

- Initialize n-by-n arrays $D^{(k)}$ for k = 0,…,n
  - $D^{(k)}[u,u] = 0$ for all u, for all k
  - $D^{(k)}[u,v] = \infty$ for all $u \neq v$, for all k
  - $D^{(0)}[u,v]$ = weight(u,v) for all (u,v) in E.
- **For** k = 1, …, n:
  - **For** pairs u,v in V^2:
    - $D^{(k)}[u,v] = \min\{ D^{(k-1)}[u,v], D^{(k-1)}[u,k] + D^{(k-1)}[k,v] \}$
- **Return** $D^{(n)}$

The base case checks out: the only path through zero other vertices are edges directly from u to v.

This is a bottom-up ***Dynamic programming*** algorithm.

We've basically just shown

- Theorem:

  If there are no negative cycles in a weighted directed graph G, then the Floyd-Warshall algorithm, running on G, returns a matrix $D^{(n)}$ so that:

  $D^{(n)}[u,v]$ = distance between u and v in G.

- Running time: $O(n^3)$
  - Better than running BF n times!
  - Not really better than running Dijkstra n times.
    - But it's simpler to implement and handles negative weights.
- Storage:
  - Enough to hold **two** n-by-n arrays, and the original graph.

Work out the details of the proof! (Or see Lecture Notes 12 for a few more details).

As with Bellman-Ford, we don't really need to store all n of the $D^{(k)}$.

What if there *are* negative cycles?

- Just like Bellman-Ford, Floyd-Warshall can detect negative cycles.
- If there is a negative cycle, then there is a path from v to v that goes through all n vertices that has cost < 0.
  - That's just the definition of a negative cycle.
- So $D^{(n)}[v,v] < 0$.
- So check for that at the end.
  - if there is such a v, return `negative cycle.`

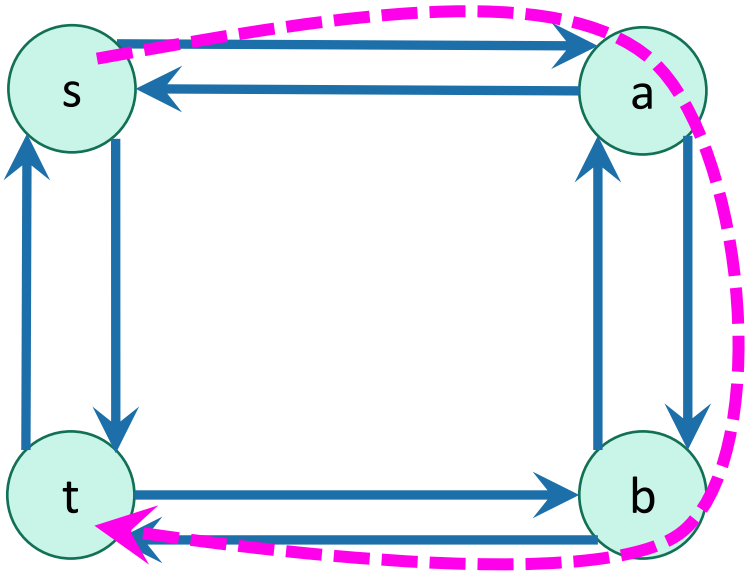
What have we learned?

- The Floyd-Warshall algorithm is another example of dynamic programming.

- It computes All Pairs Shortest Paths in a directed weighted graph in time $O(n^3)$.

Another Example?

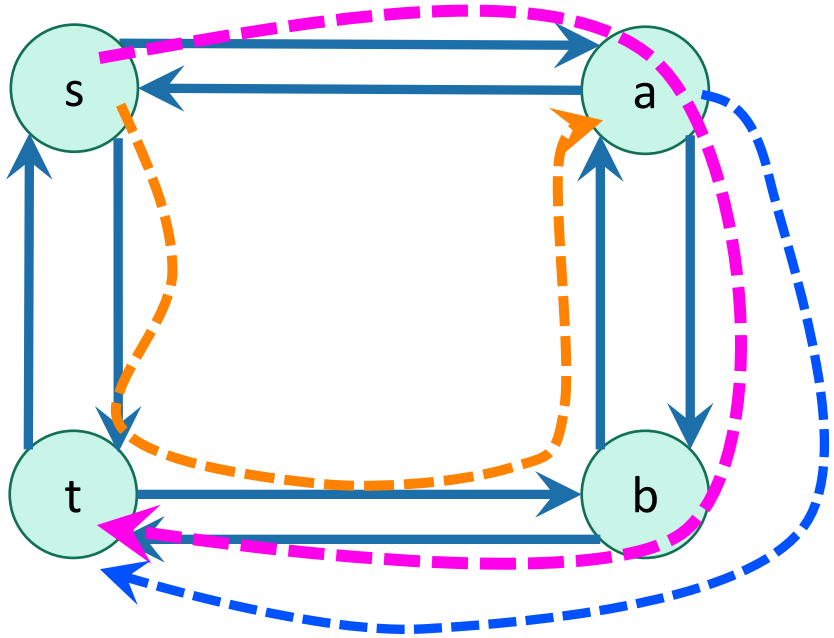
- Longest simple path (say all edge weights are 1):



What is the longest simple path from s to t?

This is an optimization problem…

- Can we use Dynamic Programming?
- Optimal Substructure?
  - Longest path from s to t = longest path from s to a
    + longest path from a to t?



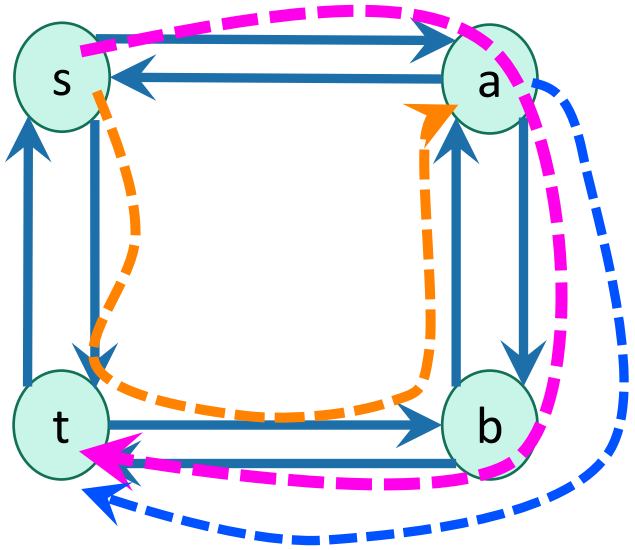
NOPE!

This doesn't work

What went wrong?

- The subproblems we came up with aren't independent:
  - Once we've chosen the longest path from a to t
    - which uses b,
  - our longest path from s to a shouldn't be allowed to use b
    - since b was already used.

- Actually, the longest simple path problem is NP-complete.
  - We don't know of any polynomial-time algorithms for it, DP or otherwise!



Recap

- Two more shortest-path algorithms:
  - Bellman-Ford for single-source shortest path
  - Floyd-Warshall for all-pairs shortest path
- ***Dynamic programming!***
  - This is a fancy name for:
    - Break up an optimization problem into smaller problems
    - The optimal solutions to the sub-problems should be sub-solutions to the original problem.
    - Build the optimal solution iteratively by filling in a table of sub-solutions.
    - Take advantage of overlapping sub-problems!

Next time

- More examples of dynamic programming!

We will stop bullets with our action-packed coding skills, and also maybe find longest common subsequences.